Physics 2660

Lecture 6: C – Part 5

Today
- Arrays and pointers
- Strings
- User-defined data structures
- Histograms
- Libraries
- Tips (etc)

1

Remember: Mid-term coming up after Spring Break! The last part of this lecture will be sample questions from the test.

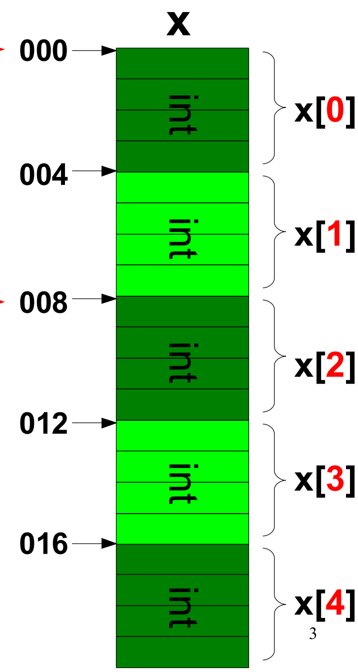
Part 1: Arrays and Pointers

2

Today we'll take another look at arrays and pointers, and how the two interact.

Arrays and Pointers:

```
int x[5];
int *top;
int *two;

top = x;
two = &x[2];
```



The name of an array can be used just like a pointer that points to the beginning of the array.

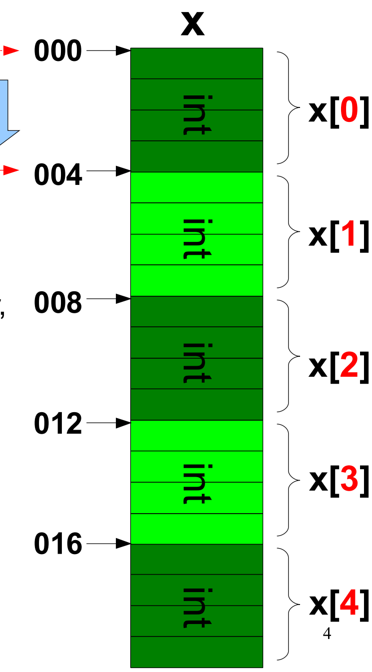
Here we define another pointer (“top”), and make it point to the beginning of the array, too.

As usual, we can get the address of individual elements using the & operator.

When you give the program an array index, the program multipies the index times the size of each element to find the address where a particular element lives.

Incrementing Pointers:

```
int x[5];
int *top;
top = x;
top++;
```



When you use the "++" operator on a pointer, it moves the address of the pointer forward by an amount equal to the size of the type of variable ("int", in this case).

Here, the pointer is initially pointing at the top of the array, so "++" moves it to the next array element.

This is our first look at "pointer arithmetic".

Pointers "know" how big the data elements are, so they "jump" to the next element when incremented.

This is why pointer type must generally match data type.

Passing Arrays as Pointers:

We can pass an array to a function by just giving the function a pointer to the beginning of the array:

```
int main () {
   const int SIZE=100;
   double d_ary[SIZE];

   reset_data( d_ary, SIZE );
....
}

void reset_data(double *data, int n) {
   int i;
   for (i=0; i<n ; i++){
      *data = 0;
      data++;
   }
}
```

By giving the name, we pass the address of the top of the array.

Array names are just pointers.

Set the data at this address to zero.

Jump to the next array element..

Again: the name of the array just acts like a pointer to the beginning of the array. This is why we can stick “d_ary” into the first argument of “reset_data”, even though “reset_data” says it wants a pointer there.

Equivalence of Pointer and Array Notation:

The two functions below do exactly the same thing, they just say it in different ways. You can freely refer to array elements by either using pointers or array (square bracket) notation.

Pointer notation:

```
void clear_data(double *data, int n) {
   int i;
   for (i=0; i<n ; i++){
     *data = 0;
      data++;
   }
}
```

Use de-referencing (* operator) and pointer arithmetic.

Array notation:

```
void clear_data(double *data, int n) {
   int i;
   for (i=0; i<n ; i++){
      data[i] = 0;
   }
}
```

Use array notation to access array elements.

The two functions above are exactly equivalent. You could drop either one into a program and it would behave just the same. The only difference is the notation.

Pointer Arithmetic:

Pointers can be manipulated with all types of integer operations. The following (and more) are all valid:

```
for (i=0; i<n ; i++){
     *data = 0;
     data++;
}

for (i=0; i<n ; i++){
     *data = 0;
     data--;
}

for (i=0; i<n/2 ; i++){
     *data = 0;
     data+=2;
}
```

Step forward.

Step backward.

Step over every 2nd element

It's up to you to make sure that you don't let your pointers go wild and point to an incorrect memory location for your application.

Be extra careful and clear when designing code to use pointers, since subtle problems with pointers can be very hard to track down.

Storage of 2-D Arrays:

```
type array[NR][NC];
```

Column

Row

[0][0]	[0][1]	...	[0][NC-1]
[1][0]	[1][1]	...	[1][NC-1]
...	...	...	...
[NR-1][0]	[NR-1][1]	...	[NR-1][NC-1]

In C, arrays are stored with “row-first” in memory. You can think of a 2-D array as an NCOLUMN array repeated NROW times.

2-D Arrays in Memory:

Column

Row

[0][0]	[0][1]	...	[0][NC-1]
[1][0]	[1][1]	...	[1][NC-1]
...	...	...	...
[NR-1][0]	[NR-1][1]	...	[NR-1][NC-1]

It is convenient to think of a 2-D array as a matrix like the one drawn above. However, all data must be stored in a linear manner in memory:

```
[0][0]      [0][1]      ...     [0][NC-1]
[1][0]      [1][1]      ...     [1][NC-1]
  ...         ...       ...       ...
[NR][0]     [NR][1]     ...     [NC-1][NC-1]
```

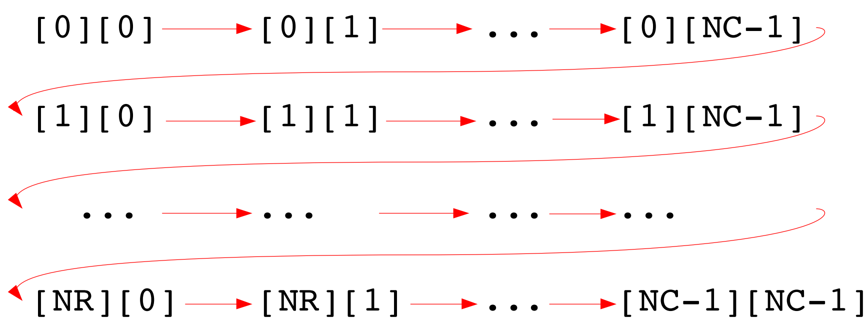

A location in memory just has one address, so pointers can't refer to an individual element of a 2-D array by its two coordinates. Instead, we need to know how many bytes we need jump from the top of the array to get to the element we're interested in.

2-D Arrays and Pointer Arithmetic:

```
int array[NR][NC];
int *array_p = array;
array_p++;
```

Equivalent to:

```
array[n][m+1];
```



Incrementing the pointer traverses the array as shown above. This is equivalent to incrementing the second array index.

Incrementing by Row:

```
int array[NR][NC];
int *array_p = array;
array_p += NC;
```

Equivalent to:

`array[n+1][m];`

[0][0]	[0][1]	...	[0][NC-1]
[1][0]	[1][1]	...	[1][NC-1]
...	...	...	...
[NR][0]	[NR][1]	...	[NC-1][NC-1]

If we want to jump all the way down to the next row, we need to increment the pointer by as many elements as there are columns. This is equivalent to incrementing the first index of a 2-D array.

More Pointer Artithmetic on 2-D Arrays:

Once we know how 2-D arrays are stored in memory, we can use pointer arithmetic to point to any array element we want:

```
// point to array [0][5]:
array_p2 = array_p + 5;

// point to array [1][3]:
array_p2 = array_p + (1*NC) + 3;

// point to last element:
array_p2 = array_p + (NR-1)*NC + NC-1;

// or, equivalently:
array_p2 = array_p + NR*NC - 1;
```

1-D Notation for 2-D Arrays:

C doesn't know the dimensions of the array that a pointer is pointing at, so we can act as though we're pointing at a 1-D array even if we originally defined a 2-D array. We just enclose the total offset in square brackets:

```
int a[NR][NC];
int *array_p = a;

// point to array [0][5]:
array_p2 = array_p[5];

// point to array [1][3]:
array_p2 = array_p[(1*NC) + 3];

// point to last element:
array_p2 = array_p[(NR-1)*NC + NC-1];
// or:
array_p2 = array_p[NR*NC - 1];
```


All that matters is the total size (total number of elements) in the array. C doesn't care how we divide them up, but we need to make sure we don't try to use elements past the end of the memory that we've reserved for our array.

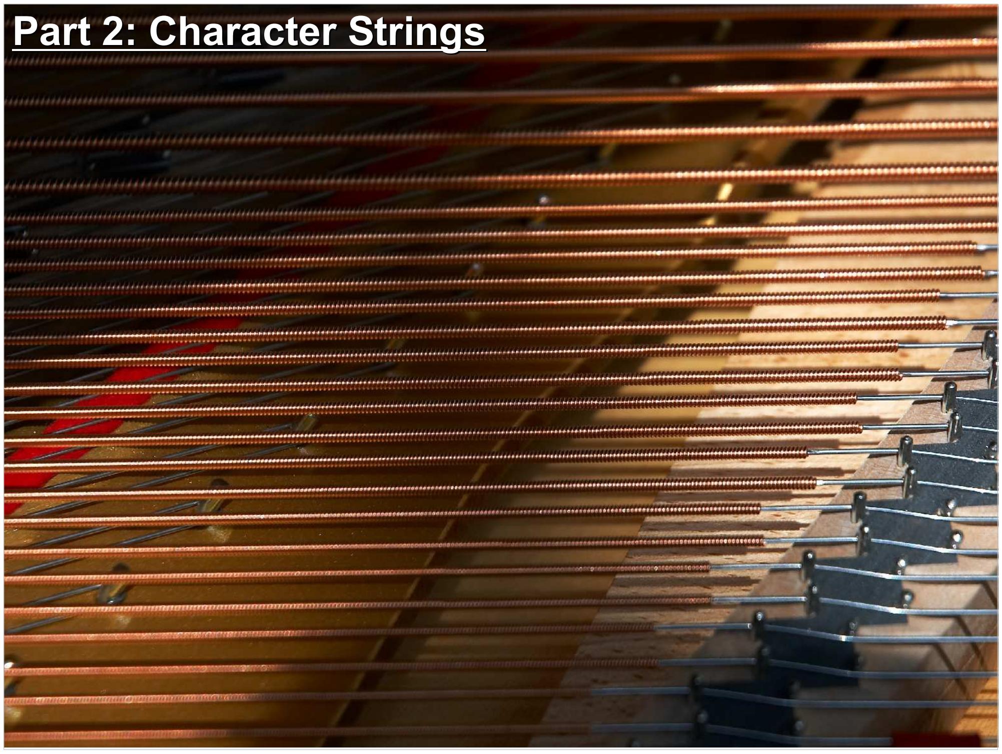
All Arrays are Linear in Memory:

C doesn't know the dimensionality of the array being pointed to. We only need to care about the total number of elements. It doesn't matter whether the array is [30], [2][15] or [2][3][5]. Each has 30 elements, and the function below could be used to clear each of them.

```
int main () {
...
  int *array_p = array;  // point to array [0][0]
  iclear(array_p, NROW*NCOL);
...
}

// This function clears any size/dimension
// integer array:
void iclear(int *pntr, int size){
  int i;
  for (i=0; i<size; i++){
    *(pntr+i) = 0;
  }
}
```

Total number of elements.

As far as C is concerned, an array is just a chunk of memory. How we subdivide it is up to us.



As we've noted before, character strings are just arrays of characters.

The strlen Function:

Character strings are just arrays of characters. The strlen function (defined in string.h) returns the length of a string.

```
#include <string.h>
...
char name[15] = "fred";
char day[] = "Tuesday";
printf("%d %d %s\n",
   sizeof(name),
   strlen(name),
   name);
printf("%d %d %s\n",
   sizeof(day),
   strlen(day),
   day);
```

Define a character string of up to 15 characters.

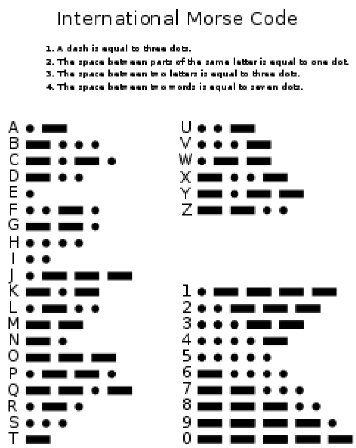
Let the compiler figure out the size.

```
15   4   fred        -> 15 bytes, 4 characters
 8   7   Tuesday     -> 8 bytes, 7 characters   (so what's up?)
```

Why does “day” have a length of 8, even though it only contains seven characters? To understand, let's digress a little and look at how computers store characters.

Character Encoding:

1840s:



1963:

American Standard Code for Information Interchange (ASCII)

1000001 A	1010101 U
1000010 B	1010110 V
1000011 C	1010111 W
1000100 D	1011000 X
1000101 E	1011001 Y
1000110 F	1011010 Z
1000111 G	
1001000 H	0110001 1
1001001 I	0110010 2
1001010 J	0110011 3
1001011 K	0110100 4
1001100 L	0110101 5
1001101 M	0110110 6
1001110 N	0110111 7
1001111 O	0111000 8
1010000 P	0111001 9
1010001 Q	
1010010 R	
1010011 S	
1010100 T	

17

00000000 = “NUL”

Prior to the 1960s, the most widespread way of communicating data electronically was morse code. When a telegram was sent, its text was encoded in morse code and transmitted through air or a wire to its destination, where it was decoded back into text.

Morse code was fine for human telegraphers, but it was clumsy for computers. In the 1960s the “American Standards Association” published a new, more computer-friendly way of transmitting text. This was called the American Standard Code for Information Interchange (ASCII).

In ASCII, each character is represented by 8 bits of information (1 byte). When you store text in a file on disk, the text is stored as ASCII characters. ASCII characters are also the way communications between a terminal (or pseudo-terminal) and a computer are encoded.

(Actually, other encodings like UTF-8 may be used these days, but the principle is the same. For simplicity, let's just assume everything is ASCII.)

Null-Terminated Strings:

```
char day[] = "Tuesday";
```

Each character takes up one byte (8 bits) in memory. A character string is just an array of characters.

But, as we've seen, C doesn't know how long an array is. When we make a statement like:

```
printf ("%s", day);
```

how does printf find the end of the string? We haven't told it the string's length explicitly.

	day
T	01010100
u	01110101
e	01100101
s	01110011
d	01100100
a	01100001
y	01111001
\0	00000000

The answer is that, in C, strings are "null-terminated".

By this we mean that a special character ("NUL") appears as the last character in the string. Because of this, functions like printf can find the end of the string by looking for the NUL.

This means that the array needs to have room for one more character than the text we're putting into it.

The special character "NUL" is a non-printable character that's represented by a string of eight zeros in memory. We sometimes write it as "\0".

Comparing Strings with strcmp:

The strcmp function (also defined in string.h) compares two strings:

```
int strcmp(char* S1, char *S2);
```

Returns:

0	if S1 = S2
>0	if S1 > S2
<0	if S1 < S2

```
#include <string.h>
int main () {
  char string1="abcde";
  char string2="fghij";
  if ( !strcmp(string1,string2) ) {
   printf ("They match.\n");
  } else {
   printf ("They don't match.\n");
  }
}
```


strcmp compares strings “lexicographically” (i.e., in dictionary order). One string is “greater than” another if it would come later in the dictionary.

Part 3: Structures

Loops are probably the most useful feature of C. My choice for the second most useful feature is structures. Let's take a look at what they're good for.

Parallel Arrays:

Let's say we wanted to store some census information about each of the fifty states.

There are several interesting facts about each state, but we can only store one fact in each variable. So we might choose to store the data in a bunch of parallel arrays, like this:

```
int    population[50];
double income[50];
double area[50];
double birthrate[50];
double deathrate[50];
```

This will work, but it's a little awkward. It would be nicer if we could bundle together all of the facts about a given state into one package.

Structures:

In addition to the regular variable types like “int” and “double”, C lets us define our own custom-made types for variables, and pack multiple pieces of data into them.

For example, we could define a 50-element array called “state” that would hold all of our census data:

```
struct {
  int population;
  double income;
  double area;
  double birthrate;
  double deathrate;
} state[50];
```

“state” is of type “struct” (a data structure), and each element of the array will contain several pieces of related data.

Note that the text ties together “struct” and “typedef”, but they're really separate things. In these notes I'm going to de-couple them so you can see what they do separately. We'll talk about typedef a little later.

The . Operator:

You can refer to a particular piece of data in a struct by using the dot operator ("."):

```
struct {
  int population;
  double income; // Average/person/year.
  double area;   // In sq. miles.
  double birthrate;  // Per year.
  double deathrate;  // Per year.
} state[50];

state[0].population = 1234567;
state[0].income = 40280.0;
state[0].birthrate = 1280.5;
state[0].deathrate = 1280.1;
```

Re-using Structs:

What if we wanted to use the same data structure for other variables? Say, for example, we wanted to store census data for a group of 100 countries. We could just re-type the struct definition:

```
struct {
  int population;
  double income; // Average/person/year.
  double area;   // In sq. miles.
  double birthrate;  // Per year.
  double deathrate;  // Per year.
} state[50];

struct {
  int population;
  double income; // Average/person/year.
  double area;   // In sq. miles.
  double birthrate;  // Per year.
  double deathrate;  // Per year.
} country[100];
```


But that would be tedious, and if we needed to change one struct later we'd probably want to remember to change the other one too. There's a better way.

Using typedef:

Instead of re-typing the struct, we could use typedef to define an alias for this struct:

```
typedef struct {
  int population;
  double income; // Average/person/year.
  double area;   // In sq. miles.
  double birthrate;  // Per year.
  double deathrate;  // Per year.
} census;

census state[50];
census country[100];
```

With typedef we've created a new variable type called "census" and now we can use this to define variables, just like "int" or "double".

25

Remember, we don't have to use typedef and struct together. We can use them separately if we want to. Next we'll see how we can use typedef on its own.

More typedef Examples:

You don't need to use struct to use typedef. You can use typedef to define aliases for any variable type:

```
//Define aliases for some types:
typedef double funds;
typedef double weight;
typedef int days;

//Use these aliases to define some variables:
funds bank_balance;
weight fish_per_month[12];
days til_christmas;
```

This may make it easier for you to re-define your variables later on. Say, for example, that you've made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you'll only need to change one line: the typedef statement that defines “funds”.

Pointers to Structs:

We can have pointers to structs just like pointers to any other type of variable:

```
typedef struct {
  int population;
  double income; // Average/person/year.
  double area;   // In sq. miles.
  double birthrate;  // Per year.
  double deathrate;  // Per year.
} census;

census states[50];
census *sptr;

sptr = states;

printf (“Pop = %d\n”, states[0].population );
printf (“Pop = %d\n", (*sptr).population );
printf ("Pop = %d\n", sptr->population );
```

When using pointers, C gives us two ways to get data from a struct.

All three of the printf statements do exactly the same thing.

Generally, when using pointers, the “->” notation will be more readable.

Using the -> Operator:

```
typedef struct {
...
} census;

census states[50];

for (i=0;i<50;i++) {
   clear_data( &states[i] );
}
...
void clear_data( census *s ) {
   s->population = 0;
   s->income = 0;
   s->area = 0;
   s->birthrate = 0;
   s->deathrate = 0;
}
```

Here's another example showing how the "->" operator can be used to refer to variables within a structure.

This program just loops through all of the states in our array, and zeroes out the data in each state's data structure.

Complex Numbers as Structs:

```
typedef struct{
   double re, im;
} Complex;

double magnitude(Complex z) {
   return sqrt( z.re*z.re + z.im*z.im );
}

void conjugate(Complex *z) {
   z->im = -1.0*z->im;
}

int main() {
  Complex q;
  q.re = 12.;
  q.im = 23.;
  conjugate(&q);
  printf("q*=%f, %f; |q|=%f \n",
            q.re, q.im, magnitude(q));
}
```

Function to find the magnitude.

Function to convert number to its complex conjugate.

Define q as "complex".

Structs are a natural way to express complex numbers. Above, we define a new type, "Complex", which is a struct containing the real and imaginary parts of a complex number. We then define some functions that accept Complex arguments (or pointers to Complex).

Passing Pointers versus Copying Data:

Here are two different ways we could write the "magnitude" function:

Pass a copy of the complex structure to the function. Do calculation from the copied data.

```
double magnitude(complex z) {
   return sqrt( z.re*z.re + z.im*z.im );
}
```

Pass a pointer to the complex structure to the function. Do calculation from the original data.

```
double magnitude(complex *z) {
   return sqrt( z->re*z->re + z->im*z->im );
}
```

Generally more efficient: less data to move around.

When writing a function ask the question:

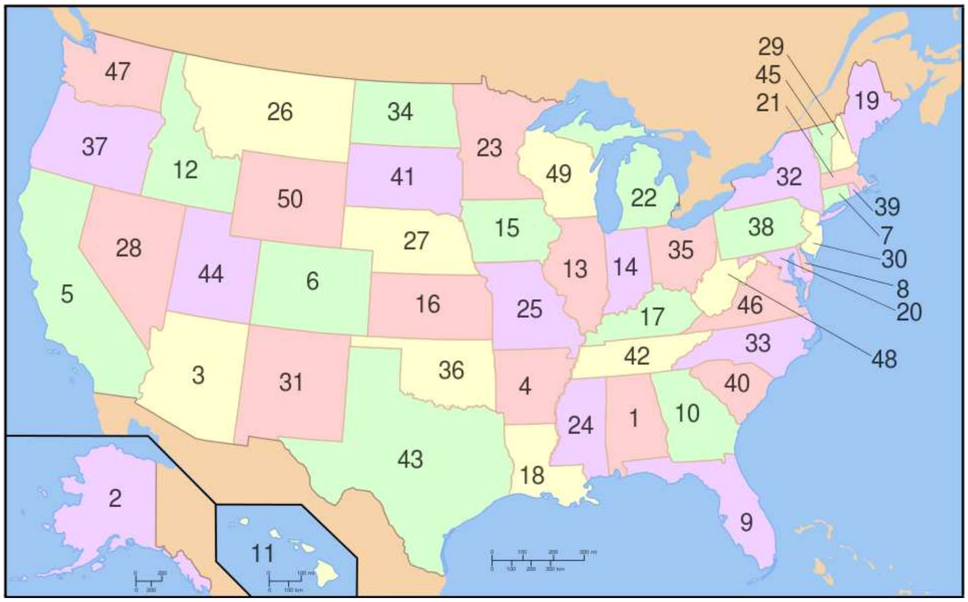
"Do I really need a new copy of the data in this function?"

If not, passing pointers can greatly increase the efficiency of your program.

Note: this makes little to no difference for small data elements like int and double. In these cases, there's no real performance justification for adding the complexity that comes with pointers.

But passing pointers should always be considered for large structures (say, a structure that includes a big array) to improve performance.

Part 4: Enumerating Lists



Names are generally easier to remember than numbers. C provides us with a rather awkward way of using names instead of integers.

The enum Statement:

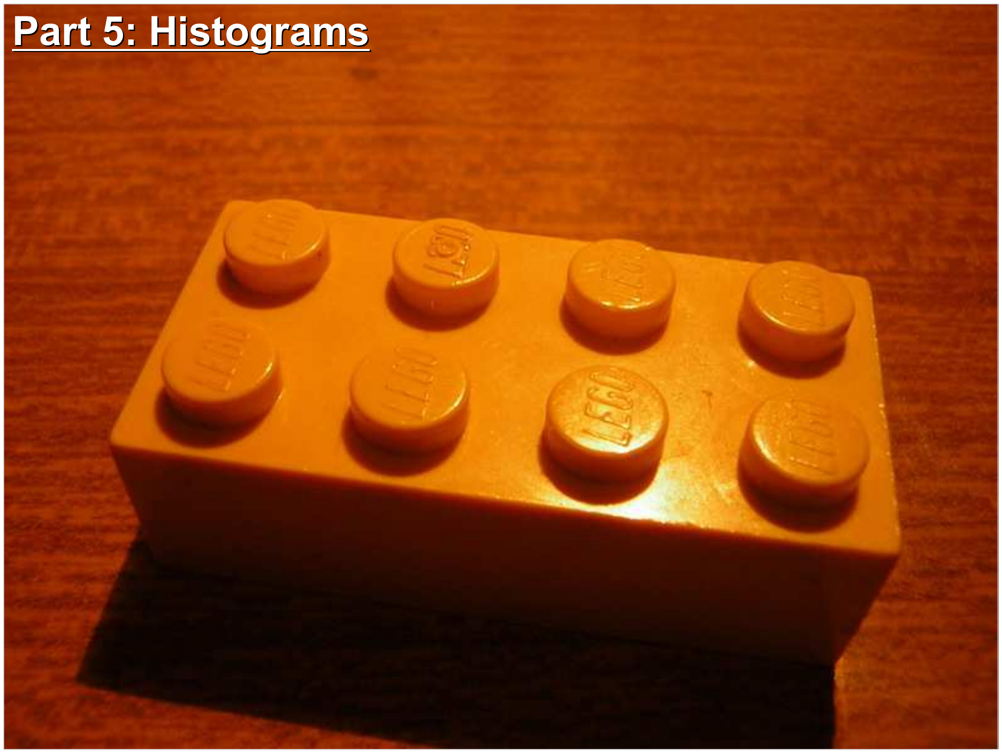
It's often more convenient to use names than numbers. In the example below, we define the variable "day" using an "enum" ("enumeration"). .

```
struct {
   double calories;
   double exercise_hours;
} data[7];

enum dayname {Sunday, Monday, Tuesday, Wednesday,
              Thursday, Friday, Saturday};
dayname day;

...

day = Sunday;
printf ("Calories on Sunday: %lf\n",
        data[day].calories);
```

An enum statement defines a list of names that will automatically be mapped to a list of integers. By default, the numbers start with zero. Enum is like typedef, in that the newly-created enum type can be used to define variables

Enum works a lot like typedef. We define a new type ("dayname", in the example above) and then we can use that type to define variables. Any variable of this type will accept any of the values we've listed in the enum.

If you were looking for the first day in the array, it would be really easy to accidentally type "1" when you meant "0". It's somewhat less likely that you'd type "Monday" instead of "Sunday". This kind of thing is the main advantage of enums.

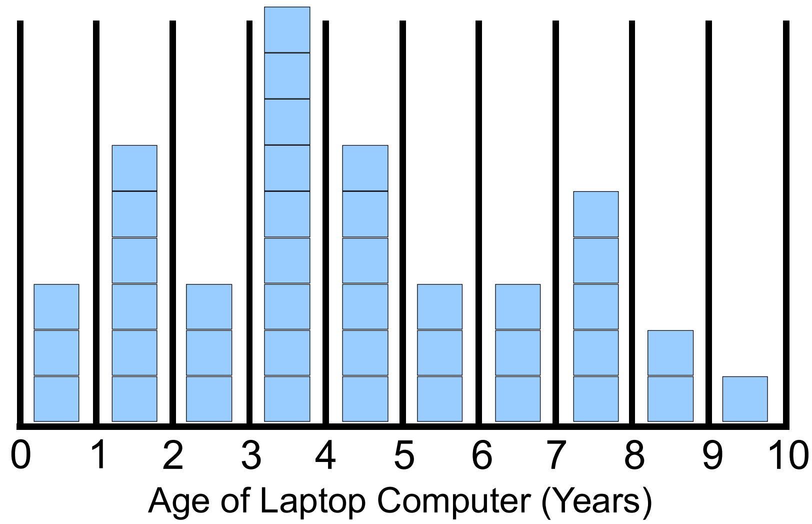


This is something we'll be looking at in lab this week.

Histograms are one of the most useful data visualization tools in Physics. If you go into research, you'll probably use them throughout your career.

What's a Histogram?

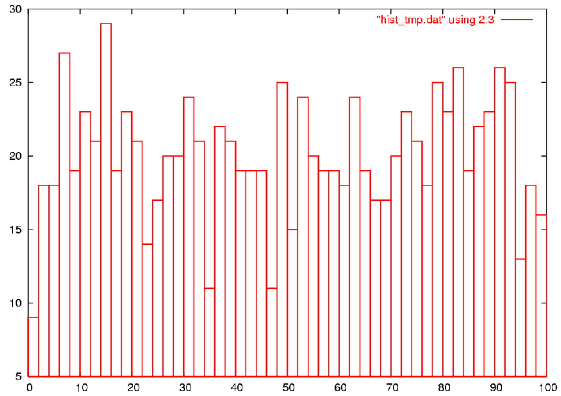
A histogram shows the distribution of data by dividing it up into discrete intervals and counting the data points that fall within each interval.



How old is your laptop computer, in years, months and days? If we looked at all of the computers in the class, it's unlikely that many of them would be exactly the same age. If we wanted to look at the distribution of ages, it would be useful to break the data into categories: say, one per year.

Reducing Data:

Histograms can reduce large quantities of data into a manageable summary.



Here we've divided the range from 0 to 100 into 50 bins.

We then generated 1000 random numbers and counted how many fell into each bin.

So, with just 50 numbers (the counts in each bin) we have a summary that shows us the distribution of our 1000 random numbers. We could go on and generate millions of numbers, and we'd still be able to summarize them in the same histogram, using only 50 counters.

A Simple struct to Store a Histogram:

Here's a structure we could use to store histogram data. At the heart of it is an array of counts.

```
#define HBINS 50

typedef struct {
  double h_array[HBINS];
  double xmin, xmax;
  int entries;
  int under_flow, over_flow;
  double sumx, sumx2;

} h1;
```

These histograms will have a fixed number of bins.

Array of bin counts.

min/max of x range.

Total # of counts.

Counts outside range.

Sums of values and squares of values, for calculating statistics.

This is the kind of data structure that you might use to store histogram data. In lab this week, we'll introduce a library of histogram functions that use this structure.

Some Basic Histogram Operations:

- create / initialize: Set range for histogram, etc.
- reset: Clear bin contents to 0.
- fill: Add a data value to the histogram.
- dump: Print contents of histogram.
- plot: Graphically display the histogram.

Function Prototypes for Simple Histogram Tools:

```
/* initialize hist. Set min/max limits for the histogram */
void h1init(h1 *hist, double xmin, double xmax);

/* add a data point to a histogram */
void h1fill(h1 *hist, double x);

/* dumps hist to screen (filename=“”) or to a file “filename”.  Returns 0 for success, 1
for error */
int h1dump(h1 *hist, char *filename);

/* calculate and return statistics for a histogram
   input: h1 *hist
   output: int *entries, double *mean, double *std_dev */
void h1stats(h1 *hist, int *entries, double *mean,
            double *std_dev);

/* plot a histogram to the screen (filename=“”) or a graphics file “filename” */
void h1plot(h1 *hist, char *filename);
```


Here are the function prototypes for some of the functions in the histogram library we'll use in lab.

Reset and initialize:

```
void h1reset(h1 *hist){
  int i;
  hist->entries=0;
  hist->sumx=0;
  hist->sumx2=0;
  hist->over_flow=0;
  hist->under_flow=0;
  for (i=0; i<HBINS; i++) hist->h_array[i]=0;
}

void h1init(h1 *hist, double min, double max){
  hist->xmax = max;
  hist->xmin = min;
  h1reset(hist);    // clear all storage variables
}
```


And here's the code for a couple of them.

Filling:

```
void h1fill(h1 *hist, double x){
  int bin=0;
  double binsize, lowedge;

  if (x < hist->min) hist->under_flow++;
  else if (x >= hist->max) hist->over_flow++;
  else {
    binsize = (hist->max - hist->min) / HBINS;
    lowedge = hist->min; // low edge of 1st bin
    while (fabs(x-lowedge) > binsize) {
      bin++;
      lowedge += binsize; // move to next bin
    }
    hist->h_array[bin]++; //increment the appropriate bin
  }
  hist->entries++;
  hist->sumx += x;
  hist->sumx2 += x*x;
}
```

Part 6: Libraries

We've been using libraries all along. C's standard libraries include all of the I/O functions, math functions, random number generation functions and so forth that most of our programs have relied upon. Now we'll start looking at how you can create your own libraries, containing your own functions.

Building Libraries:

To build a library, first make object files (as we have done before):

```
g++ -O -Wall -c hist.cpp
g++ -O -Wall -c random.cpp
```

Next combine the object files into a library:

```
ar -csr libp2660.a hist.o random.o
```

The archive command is used to create your library, the syntax we will use is:

```
ar -csr lib<name>.a file1.o file2.o ...
```

You can list the contents of your library with a command like:

```
ar -t libp2660.a
```

hist.o
random.o

csr =

c: “Create archive, if it doesn't already exist.”
s: “Add a table of contents to the archive.”
r: “Put the following files into the archive, replacing any already-existing files with the same names.”

An Example Program:

It's very easy to use functions from your library within your programs:

```
// Header files for your library
#include "random.hpp"
#include "hist.hpp"

int main(){
  h1 myHist;
  // Set range for histogram's x-axis
  h1init( &myHist, 0, 100);
  for (int i=0; i<1000; i++) {
    // Fill the histogram w/ 100 data points
    // from the function randn:
    h1fill(&myHist, randn(50,10));
  }
  // Plot the histogram to the screen
  h1plot(&myHist, "");
  return 0;
}
```


This shows a program that uses some of the histogramming functions from the "Physics 2660 library" that we'll be using soon.

Using Your Library to Build a Program:

To use your library with a program:

1) make sure your program includes header files defining the functions you use

2) tell the linker how to find your library

Let's say your program file is called test_hist.cpp. You would build the program as follows:

```
g++ -O -Wall test_hist.cpp -o test_hist -L. -lp254
```

-L specifies a new directory to search for library files (here we add ".", the current directory, to the library search path)

-l (small "L") gives the name of a library (libp254.a) to search for object files needed to complete your program. Note that the "lib"/".a" prefix/suffix is omitted from the command

Using Your Library to Build a Program:

In general the header files and libraries will not be located in your current working directory, so for more complex programs the build command could be of the form:

```
g++ -O -Wall \
-I<include_dir1> -I<include_dir2> \ test_hist.cpp
-o test_hist \
-L<lib_dir1> -L<lib_dir2>  \
-l<lib1> -l<lib2> -l<lib3>
```

In this week's lab we'll practice making and using code libraries and start using a simple histogram library to visualize data generated in our programs.

Mid-Term Example Problems:

We have a mid-term exam coming up after Spring break. The following are some example questions from previous mid-terms. (Some of them may even show up this time!) I'll try to make the real questions no harder than these, and I'll avoid unneccessarily tricky questions.

There will be some more example questions included with this week's homework assignment.

Most of the problems on the mid-term will be multiple-choice, with a few short-answer questions.

The Shell:

Write out or choose the GNU/Linux shell command that does the following:	
1) lists files in your directory	`ls`

2) lists files in your directory, with sizes shown	`ls -l`
	or
	`ls -al`

3) renames the file ***my.dat*** to ***your.dat***	`mv my.dat your.dat`
4) places file ***a.txt*** in a subdirectory called ***sub***	a) rn a1.txt a2.txt b) mv a.txt c) rename a.txt sub/a.txt d) mv a.txt sub

7

Answers are in red.

Compiling and Linking:

Write out or choose the GNU/Linux shell command that does the following:	
1) Makes the executable file ***code*** from ***code.cpp***	a) g++ -c code.cpp b) ar -csr code.cpp c) g++ -o code code.cpp d) g++ code.cpp > code
2) Creates an object file from ***code.cpp***	a) g++ -c code.cpp b) ar -csr code.cpp c) g++ -o code code.cpp d) g++ code.cpp > code
3) Compiles ***code.cpp*** with warnings and optimization turned on	a) g++ -c code.cpp b) g++ -O code -Wall -o code.cpp c) g++ -Wall code -O -o code.cpp d) g++ -Wall -O -o code code.cpp
4) Compiles ***code.cpp*** and links with an object file to make an executable	a) g++ -o code code.cpp mylib.o b) g++ -O mylib.o code.cpp c) g++ -Wall -L mylib.o code.cpp d) g++ -Wall -O -o code code.cpp

1. "-o code" means "write the output executable into the file "code".

2. The "-c" flag produces an object file.

3. The "-Wall" flag turns on warnings, and the "-O" (capitol O) flag turns on optimization.

4. To link with an object file, just add the name of the object file to the end of the command.

The C Language (1):

Choose the best answer.	

1) Define an integer variable, ***i***:	`a) int i;`
	`b) int &i;`
	`c) integer i;`
	`d) int *i;`

2) Define a floating point variable with value=3.14 whose value cannot be changed:	`a) #define PI=3.14;`
	`b) const double PI=3.14;`
	`c) #define PI 3.14`
	`d) static float PI=3.14;`

3) The statement to read a double value into the variable named ***discount*** is:	`a) scanf("%lf", discount);`
	`b) scanf("%d", &discount;`
	`c) scanf(discount);`
	`d) scanf("%lf", &discount);`

4) Print the double variable ***q*** in scientific or floating point notation, whever is more compact:	`a) printf("%ef", q);`
	`b) printf("%e",  q);`
	`c) printf("%g",  q);`
	`d) printf("%lf", q);`

49

1. Well, obviously.

2. Why didn't we use one of the “#define” statements? In part because these don't define a floating-point variable. They just specify some text that we'd like to find-and-replace in our program. The “#define” statements don't tell the compiler anything about the type of data. Also, why “const”? Because this tells the compiler that the value of this variable can't be changed. (Don't confuse this with “static”, which means something else entirely.)

3. Note the %lf, for type “double”, and the “&”.

4. The “%g” format specifier does what we want.

The C Language (2):

Choose the best answer.	
1) Using the file pointer, ***input_file***, open the file ***results.dat*** for read mode.	a) `openf(“results.dat”,”r”,input_file);` b) `open(input_file,“results.dat”,“r”);` c) `fopen( input_file, “results.dat”, “r” );` d) `input_file = fopen(“results.dat”, “r” );`

2) Which code snippet reads an integer from the program's command line?	`int main(int argc, char *argv[]){`
	`...`
	a) `int i = argv[1];`
	b) `int i = atoi(argv[1]);`
	c) `int i = atoi(argv[0]);`
	d) `int i = (int)argv[1]);`

3) Function pointer type that can point to sqrt() function in the C math library:	a) `double (*f) (double x)` b) `double *f(double x)` c) `double &f(double x)` d) `double *f(double x)`

1. This is the right form for the “fopen” function call.

2. Why not “c”? Because argv[0] is the name of the program, not the first command-line argument. Why “atoi”? Because argv[1] is a character string, not an integer. The atoi function converts strings into integers.

3. The sqrt function takes one “double” argument, and returns a double. That's what this function pointer says.

The C Language (3):

Choose the best answer.	

1) Sum all multiples of 17 between 33 and 123456:	**a)**
	`for (int i = 33; i<123456; i++){`
	`  sum += (i/17) * (i%17);`
	`}`
	
	b)
	`for (int i = 33; i<123456; i++2){`
	`  if (i%17) sum += i;`
	`}`
	
	c)
	`for (int i = 33; i<123456; i++){`
	`  if (!(i%17)) sum += i;`
	`}`
	
	d)
	`for (int i = 33; i<123456; i+17){`
	`  sum += i;`
	`}`

To find multiples of 17, we can use the modulo operator (%). This returns the remainder. So, if the quantity i%17 is zero, that means that i is a multiple of 17. When i%17 is zero, "!(i%17)" is 1.

The C Language (4):

Choose the best answer.	
1) Which of the following gives the memory address of integer variable a?	a) `*a;` b) `a;` c) `&a;` d) `address(a);`
2) Which of the following gives the value stored at the address pointed to by pointer a?	a) `a;` b) `val(a);` c) `*a;` d) `&a;`
3) Which of the following gives the size, in bytes, of an “int” variable?	a) `*int;` b) `sizeof(int);` c) `strlen(int);` d) `SIZE(int);`
4) Which is a valid typecast?	a) `i(double);` b) `double:i;` c) `to (double,i);` d) `(double)i;`

1. The ampersand (&) returns the address of a variable.

2. The star (*) returns the data stored at this address.

3. Sizeof returns the size, in bytes, of a variable or expression.

4. The others aren't valid C statements.

The C Language (5):

Choose the best answer.	
1) What is the only function all C programs must contain?	a) `start()` b) `system()` c) `main()` d) `program()`
2) Which of the following is the correct operator to compare two numerical variables?	a) `:=` b) `=` c) `equal` d) `==`
3) How many times is a “do while” loop guaranteed to loop?	a) `0` b) `Infinitely` c) `1` d) `Variable`
4) Evaluate: `!(1 && !(0 || 1)).`	a) `True` b) `False` c) `Unevaluatable`

1. You knew that.
2. Remember: “==” compares two things, but “=” assigns one thing to another. And for string comparisons, we need to use “strcmp”.
3. “do while” loops always execute at least once, but a “while” loop won't necessarily ever be executed.
4. Starting from the innermost parentheses:

   0 || 1 is an “or”, so it's true if either one is true.
   --> 1

   !1 is the logical opposite of 1, so:
   --> 0

   1 && 0 is an “and”, so it's only true if both are.
   --> 0

   !0 is the logical opposite of 0, so:
   --> 1, or True.

The C Language (6):

Choose the best answer.	
1) If N is an integer variable with the value 10, what is the value of x after this statement? `double x = 1/N*2.0;`	a) `inf` b) `0.0` c) `.05` d) `.2`
2) Which is not a valid C statement?	a) `x = a + b;` b) `r = sqrt(x*x+y*y)` c) `val = sin(PI/n);` d) `a++;`
3) What is the index number of the last element of an array with 29 elements?	a) `29` b) `28` c) `0` d) `Programmer-defined`
4) Which of the following gives the memory address of the first element in array foo, an array with 100 elements?	a) `foo[0];` b) `foo;` c) `&foo;` d) `foo[1];`

1. C will evaluate this left-to-right, so:
   "1" is an integer and "N" is an integer, so 1/N is zero. Zero times 2.0 is zero.

2. This is missing a semicolon.
   "You must not forget the semicolon, best beloved."

3. The index of an array goes from zero to N-1.

4. The name of an array is equivalent to a pointer that points to the top of the array.

The C Language (7):

Choose the best answer.	
1) How is the ***continue*** statement used?	a) To continue to the next line of code b) To return from a functionality c) To stop the current iteration and begin the next iteration d) As an alternative to the else statement
2) Which of the following compares two strings?	a) `compare();` b) `stringcompare();` c) `cmp();` d) `==` e) `strcmp();`
3) How does one write the statement, "if i NOT equal to zero"?	a) `if ( i = !0 )` b) `if !( i == 0)` c) `if ( i != 0 )` d) `if ( i <> 0 )`

1. Compare how “continue” and “break” work.

2. We talked about this earlier in today's lecture.

3. Note that “!=” is another one of C's comparison operators, like “==”, “>”, “<”, etc. Some other languages would allow expression “b”, but C doesn't like it.

The C Language (8):

1) Given the variables:

```
int sum;
int maxint;
```

Write a statement that tests to see if ***sum*** is equal to 1000 and also that ***maxint*** is between 10 and 50, inclusive.

If the condition is satisfied, print the text "OK".

```
if ( sum == 1000 &&
     maxint >= 10 &&
     maxint <= 50 ) {
    printf ("OK\n");
}
```

or

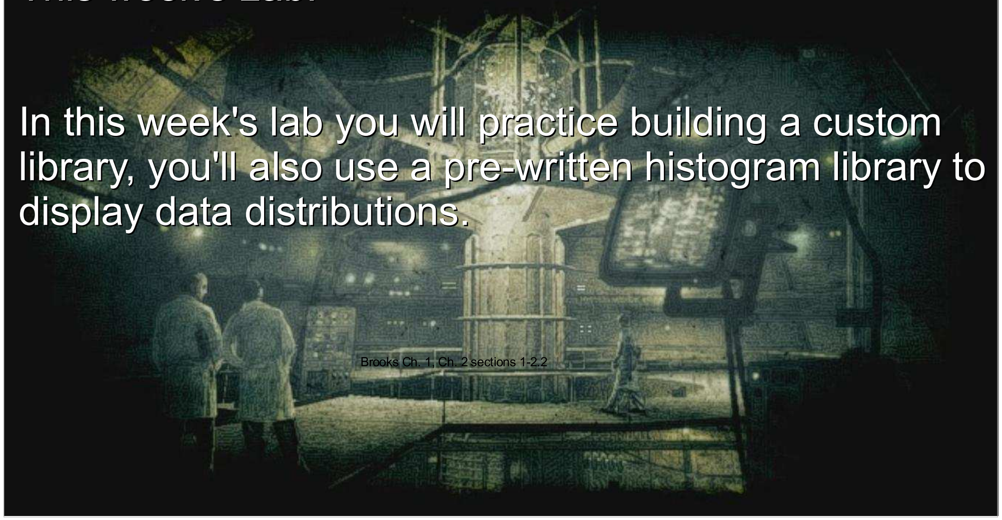
```
if ( sum == 1000 &&
     maxint >= 10 &&
     maxint <= 50 )
    printf ("OK\n");
```

Next Time:

After Spring Break: Mid-Term Exam!

This week's Lab:

In this week's lab you will practice building a custom library, you'll also use a pre-written histogram library to display data distributions.



Brooks Ch. 1, Ch. 2 sections 1-2.2

The End

58

Thanks!